CSC1101: Structured Programming

Lecture 01 (BSCS, BSDS, BSIT)

OBJECT ORIENTED PROGRAMMING: L1

Mr. Ian Raymond Osolo
(Department of Computing & Technology
Faculty of Engineering, Design & Technology

Mon 16th October 2023

Lecture Overview

- Subtopic 1
- Subtopic 2
- Subtopic 3
- Subtopic 4
- Subtopic 5

Right into it

- A **class** is a blueprint or template that defines the structure and behavior (attributes and methods) for objects in Python.
- An **object** is an instance of a class, representing a specific entity with data and behavior defined by its class.

```
class Dog:
    def __init__(self, name, breed, age):# constructor
        self.name = name
        self.breed = breed
        self.age = age

#Creating objects (instances) of the Dog class
dog1 = Dog("Buddy", "Golden Retriever", 3)
dog2 = Dog("Max", "German Shepherd", 5)

print(dog1.name)   Output: Buddy
print(dog2.breed)   Output: German Shepherd
```

OBJECTS

- Python supports many different kinds of data

```
1234      3.14159      "Hello"      [1, 5, 7, 11, 13]
{"CA": "California", "MA": "Massachusetts"}
```

- each is an **object**, and every object has:
  - a **type**
  - an internal **data representation** (primitive or composite)
  - a set of procedures for **interaction** with the object
- an object is an **instance** of a type
  - `1234` is an instance of an `int`
  - `"hello"` is an instance of a string

OBJECT ORIENTED PROGRAMMING (OOP)

- **EVERYTHING IN PYTHON IS AN OBJECT** (and has a type)
- can **create new objects** of some type
- can **manipulate objects**
- can **destroy objects**
  - explicitly using `del` or just “forget” about them
  - python system will reclaim destroyed or inaccessible objects – called “garbage collection”

WHAT ARE OBJECTS?

- objects are **a data abstraction** that captures…

(1) an **internal representation**

- through data attributes

(2) an **interface** for interacting with object

- through methods (aka procedures/functions)
- defines behaviors but hides implementation

EXAMPLE:
[1,2,3,4] has type list

- how to **manipulate** lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly

ADVANTAGES OF OOP

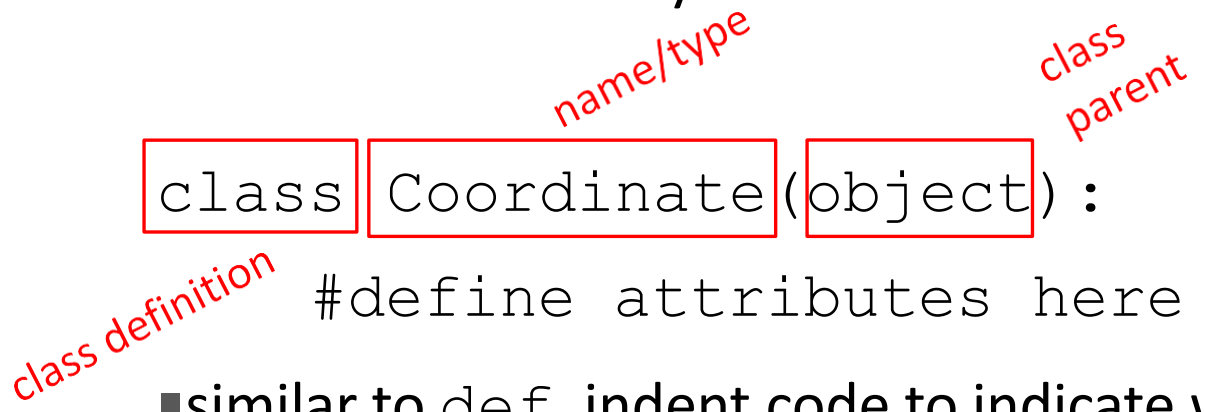
- **bundle data into packages** together with procedures that work on them through well-defined interfaces
- **divide-and-conquer** development
  - implement and test behavior of each class separately
  - increased modularity reduces complexity
- classes make it easy to **reuse** code
  - many Python modules define new classes
  - each class has a separate environment (no collision on function names)
  - inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

CREATING AND USING YOUR OWN TYPES WITH CLASSES

- make a distinction between **creating a class** and **using an instance** of the class
- **creating** the class involves
  - defining the class name
  - defining class attributes
  - *for example, someone wrote code to implement a list class*
- **using** the class involves
  - creating new **instances** of objects
  - doing operations on the instances
  - *for example,* `L=[1,2]` *and* `len(L)`

DEFINE YOUR OWN TYPES

- use the `class` keyword to define a new type



- similar to `def`, indent code to indicate which statements are part of the **class definition**
- the word `object` means that `Coordinate` is a Python object and **inherits** all its attributes (inheritance coming lecture)
  - `Coordinate` is a subclass of `object`
  - `object` is a superclass of `Coordinate`

WHAT ARE ATTRIBUTES?

- data and procedures that "**belong**" to the class
- **data attributes**
  - think of data as other objects that make up the class
  - *for example, a coordinate is made up of two numbers*
- **methods** (procedural attributes)
  - think of methods as functions that only work with this class
  - how to interact with the object
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

- first have to define **how to create an instance** of object
- use a **special method called `__init__`** to initialize some data attributes

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

special method to create an instance — is double underscore

what data initializes a Coordinate object

parameter to refer to an instance of the class

two data attributes for every Coordinate object

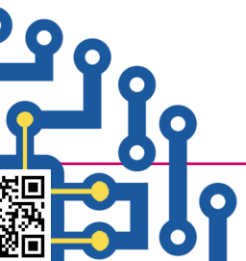
ACTUALLY CREATING AN INSTANCE OF A CLASS

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

use the dot to access an attribute of instance `c`

- data attributes of an instance are called **instance variables**
- don't provide argument for `self`, Python does this automatically



WHAT IS A METHOD?

- procedural attribute, like a **function that works only with this class**
- Python always passes the object as the first argument
  - convention is to use **`self`** as the name of the first argument of all methods
- the "**.**" **operator** is used to access any attribute
  - a data attribute of an object
  - a method of an object

DEFINE A METHOD FOR THE `Coordinate` CLASS

```
class  Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

use it to refer to any instance

another parameter to method

dot notation to access data

- other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

HOW TO USE A METHOD

```
def distance(self, other):
    # code here
```

method def

Using the class:

- conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

object to call method on

name of method

parameters not including `self` (`self` is implied to be `c`)

- equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

name of class

name of method

parameters, including an object to call the method on, representing `self`

OOP vs Procedural Programming

- Do some research on this.
- You need to know at least 5 “comparisons”

Examples

- OOP Languages: Python, Java, C++
- These languages provide support for creating objects, classes, inheritance, and more.
- Procedural Programming Languages: C, Fortran, Pascal
- These languages focus on procedural logic and functions, with limited structural modularity.

Uganda Christian University
P.O. Box 4 Mukono, Uganda
Tel: 256-312-350800
https://ucu.ac.ug/ Email: info@ucu.ac.ug.
@ugandachristianuniversity @UCUniversity
@UgandaChristianUniversity

MIT OpenCourseWare
https://ocw.mit.edu

Department of Computing & Technology
FACULTY OF ENGINEERING, DESIGN AND TECHNOLOGY
Tel: +256 (0) 312 350 863 | WhatsApp: +256 (0) 708 114 300
@ucucomputeng @ucu_ComputEng
https://cse.ucu.ac.ug/ Email: dct-info@ucu.ac.ug